Elena Martinez

Capstone: Survey Existing Research and Reproduce Available Solutions

Source 1) Predictive Analysis with Different Approaches | Kaggle

ADABOOST REGRESSOR

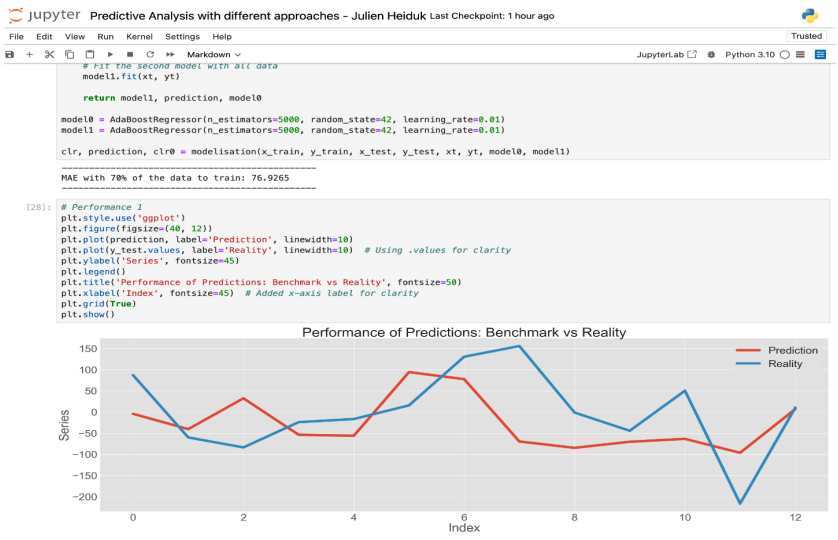
Data Preparation & Feature Engineering:

- The data is structured to support time series forecasting. It includes columns for the date, day of the week, and daily differences in visits (`diff`). The code also creates "lagged" versions of the `diff` feature, essentially creating 7 lag features (`lag1`, `lag2`, ..., `lag7`) to capture recent values for model input.
- The `diff` column is created to represent daily changes in visit counts, and the `lag_func` function generates features based on past values of `diff`, making the model aware of previous trends.

AdaBoost Regressor:

- Two AdaBoost models are initialized with 5,000 estimators, a low learning rate (0.01)
- The first AdaBoost model (`model0`) is trained on the training set and tested on the test set, with metrics like Mean Absolute Error (MAE) and \(R^2 \) score calculated to assess performance.
- The second AdaBoost model (`model1`) is then trained on the entire dataset (both training and testing portions) for final forecasting.

Prediction and Plotting:



- Generates future predictions (up to 30 days) by using the most recent lag values as inputs and iteratively predicting the next day.
- If the model predicts negative values (which are unrealistic for visit counts), these are set to zero.

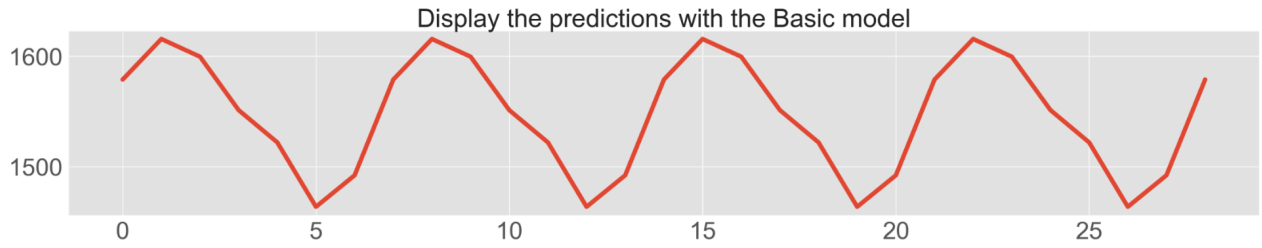
MAE: 76.9265

BASIC MODEL

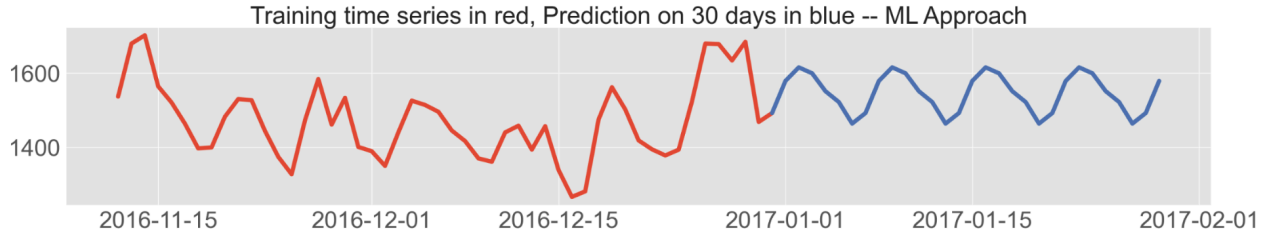
With the basic model, it simply assumes that future values for each day will follow the historical average for that specific weekday. For example, if Mondays typically had an average of 200 visits, it will predict 200 visits for each future Monday. This approach doesn't account for trends or seasonal changes but gives a simple baseline prediction.

```
[40]: plot_basic = np.array(basic_approach[basic_approach['date'] > last_date].sort_values(by='date').Visits)
```

```
[41]: plt.figure(figsize=(30, 5))
plt.plot(plot_basic, linewidth=7)
plt.title('Display the predictions with the Basic model', fontsize=40)
plt.show()
```




```
[42]: df_lagged = basic_approach[['Visits','date']].sort_values(by='date')
df_train = df_lagged[df_lagged['date'] <= last_date]
df_pred = df_lagged[df_lagged['date'] >= last_date]
plt.style.use('ggplot')
plt.figure(figsize=(30, 5))
plt.plot(df_train.date,df_train.Visits, linewidth=7)
plt.plot(df_pred.date,df_pred.Visits,color='b', linewidth=7)
plt.title('Training time series in red, Prediction on 30 days in blue -- ML Approach', fontsize=40)
plt.show()
```



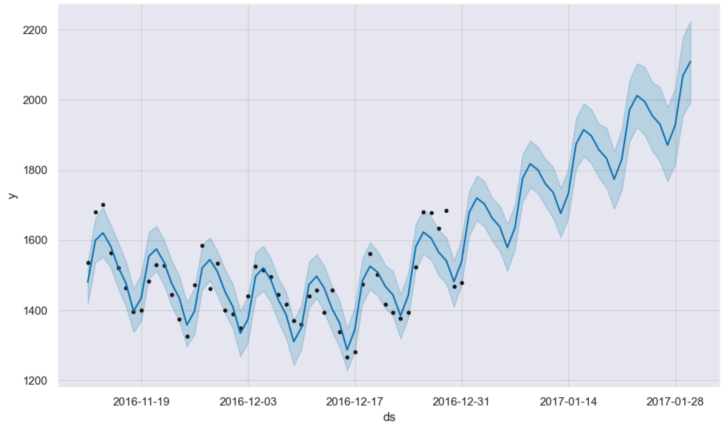
PROPHET

This simple Prophet model works by automatically detecting and modeling trends and seasonal patterns in the data, then using those patterns to forecast the next 30 days. Prophet is helpful because it handles seasonality well, making it effective for data with recurring patterns over time.

```
sns.set(font_scale=1)
df_date_index = times_series_means[['date','Visits']]
df_date_index = df_date_index.set_index('date')
df_prophet = df_date_index.copy()
df_prophet.reset_index(drop=False,inplace=True)
df_prophet.columns = ['ds','y']

m = Prophet()
m.fit(df_prophet)
future = m.make_future_dataframe(periods=30,freq='D')
forecast = m.predict(future)
fig = m.plot(forecast)
```

```
13:52:57 - cmdstanpy - INFO - Chain [1] start processing
13:52:57 - cmdstanpy - INFO - Chain [1] done processing
```



KERAS

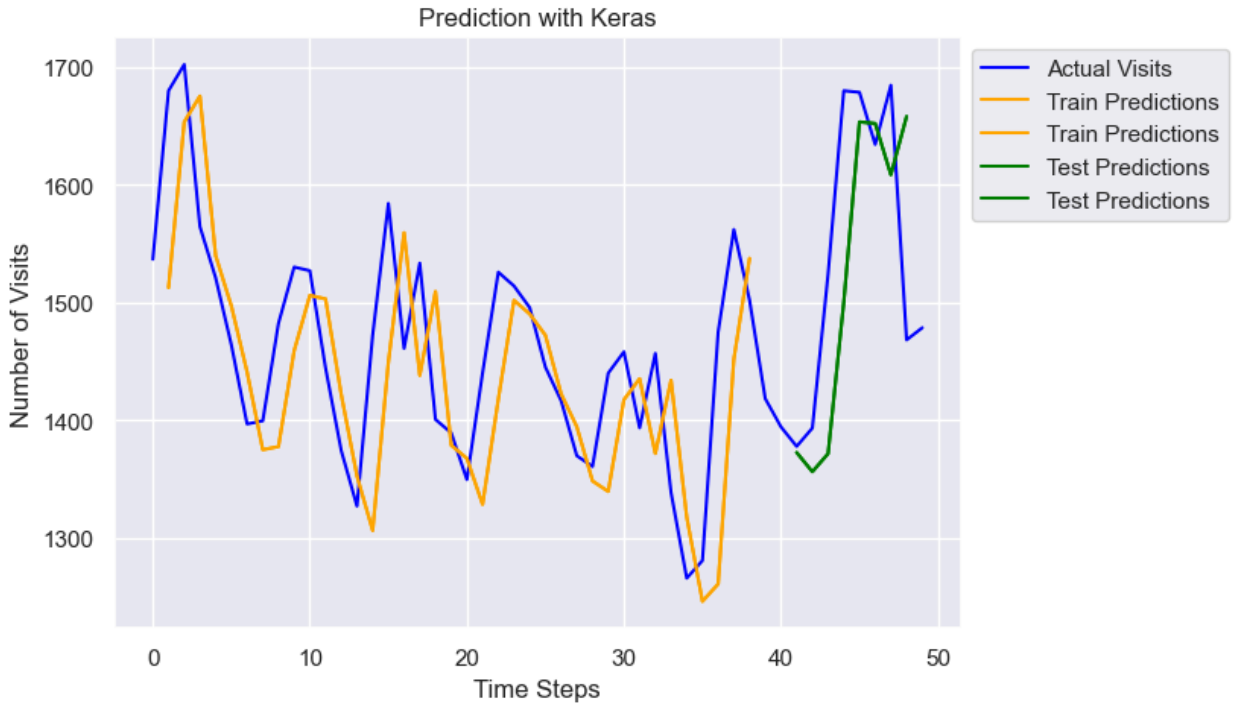
Performs time series prediction using a sequential neural network model in Keras.

Modeling & Training:

- Two dense (fully connected) layers:
- First layer, 8 neurons with ReLU activation.
- Second layer, 1 neuron to output the predicted visit count.
- Compiled with Mean Absolute Error (MAE) as the loss function and the Adam optimizer.
- Trained on the training dataset (`trainX`, `trainY`) for 150 epochs with a batch size of 2.

Train Score: 66.45 MSE

Test Score: 85.43 MSE



Source 2) Time Series Data with ARIMA Model | Kaggle

In this notebook, we performed more exploratory data analysis, including:

- Number of null values per day.
- Metadata (title, language, access type, access origin) and adding them as new columns to the DataFrame.
- Average page views for each language.
- Average & frequency of views by language over time.
- Trends for access types (e.g., all-access, mobile, desktop) for each language.

ARIMA

Then, we used an ARIMA model to forecast future page views with walk-forward validation. This is a method of splitting the data into training and test sets in a way that respects the chronological order of observations, crucial for time-dependant data.

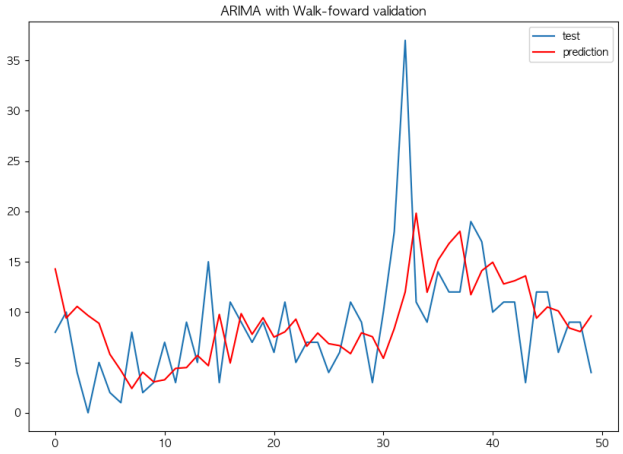
[48]:
```
from statsmodels.tsa.arima.model import ARIMA

train, test = X_train[86431], y_train[86431]
record = [x for x in train]
predictions = list()
# walk-forward validation
for t in range(len(test)):
    # fit model
    model = ARIMA(record, order=(4,1,0))
    model_fit = model.fit()
    # forecast one step
    yhat = model_fit.forecast()[0]
    # store the result
    predictions.append(yhat)
    record.append(test[t])
```

[49]:
```
from math import sqrt
from sklearn.metrics import mean_squared_error
# evaluate forecasts
rmse = sqrt(mean_squared_error(test, predictions))
print('RMSE: %.3f' % rmse)
# plot forecasts against actual outcomes
fig = plt.subplots(figsize=(10,7))
plt.plot(test)
plt.plot(predictions, color='red')
plt.legend(['test', 'prediction'])
plt.title('ARIMA with Walk-foward validation')
plt.show()
```

```
RMSE: 5.859
```

MSE: 34.328.



With walk-forward validation, we trained the model on a small segment of historical data (for example, the first few months/years). Then, tested the model on the next time step (for example, the next day, week, or month) and record the prediction accuracy. After testing, we can expand the training set by including the previously tested time step, so the model learns from all previous observations. This process is repeated—training, then testing on the next step, then expanding the training set—until the entire dataset has been used.

]:
```
train, test = X_train[86431], y_train[86431]
record = [x for x in train]
predictions = list()
# walk-forward validation
for t in range(len(test)):
    # fit model
    model = ARIMA(record, order=(1,0,5))
    model_fit = model.fit()
    # forecast one step
    yhat = model_fit.forecast()[0]
    # store the result
    predictions.append(yhat)
    record.append(test[t])

# evaluate forecasts
rmse = sqrt(mean_squared_error(test, predictions))
print('RMSE: %.3f' % rmse)
# plot forecasts against actual outcomes
fig = plt.subplots(figsize=(10,7))
plt.plot(test)
plt.plot(predictions, color='red')
plt.plot(predictions, color='red')
plt.legend(['test', 'prediction'])
plt.title('ARIMA with Walk-foward validation 2')
plt.show()
```

```
RMSE: 5.680
```

AFTER PARAMETER-TUNING:

MSE: 32.2624

The final model achieved the lowest Mean Squared Error (MSE), but it struggled to capture large spikes accurately.